REPORT FOR NASA GRANT NAG9-351

Redundancy Management for Efficient Fault Recovery in NASA's Distributed Computing System

Miroslaw Malek, Principal Investigator

Mihir Pandya

Kitty Yau

Department of Electrical and Computer Engineering,
The University of Texas at Austin,
Austin, Texas 78712.

February 15, 1991

Contents

1 INTRODUCTION

The proliferation of increasingly powerful and complex multiprocessor systems has made fault-tolerant design a necessity. Optimizing fault tolerance in multiprocessor systems is a very difficult task because it involves multi-dimensional tradeoffs. The system architecture, the computation structure, the implementation technology, the frequency, duration and location of faults, and many other factors all have certain impact on the effectiveness of a particular fault-tolerant approach. In our research, we have attempted to look at different areas of fault tolerance and have tried to integrate them under one umbrella. A comprehensive approach to fault tolerance is perhaps the only solution that may succeed in the difficult task of redundancy management. Such an approach covers design for fault tolerance and testing. A comprehensive approach requires a proper perspective, especially in distributed systems. A four layered view of fault tolerance in multiprocessor system, as shown in Figure 1.1, may prove to be very useful.

The first requirement for successful redundancy management for fault tolerance requires synchronization. Synchronization, in a broad sense, is vital in coordination and universal agreement on the events occurring in time, including faults. It is crucial in successful fault management to know the occurrence of timing events with reasonable accuracy. Once synchronization problem is solved [1], the next important problem is that of reliable communication, especially reliable broadcast. Here again several, many of them practical, reliable broadcast protocols have been proposed [2]–[5] and implemented at, for example, AT&T and GMD [6] and are being considered for IBM's Air Traffic Control distributed computing system [7]. The next problem that has been extensively studied is that of consensus on who is faulty and who is not in a distributed system. Despite an extensive research, as documented in our survey [8], practical implementation are not so common. Our comparison method [9] has been implemented at AT&T and the distributed consensus problem has been recently implemented at CMU [10] and is being considered for IBM's Air Traffic Control System [11]. Since there are still several unanswered questions in distributed consensus protocols, we are actively pursuing this problem in a large network environment that will conclude with implementation on the University of Texas Network.

Once synchronization, reliable communication and consensus are correctly implemented we may proceed with recovery or fault masking, usually followed by reconfiguration and repair. In order to mask faults, computations are replicated on different processors and replicated results are voted upon. A popular example is the N-Modular Redundancy scheme [12]. Fault masking requires massive space (hardware and software) redundancy.

The alternative to fault masking is recovery. A faulty processor may deliver incorrect computational results on time, delay them or not deliver any of them. Recovery is the process of recuperating the correct calculations a processor would have produced were it not faulty. It can be implemented through backward or forward recovery schemes. Backward recovery is more general. Its implementation is facilitated by storing checkpoints corresponding to correct states of the computation. Once faults are detected, the whole application execution is backtracked to the most recent checkpoint. The application is

A FAULT-TOLERANT COMPUTING PERSPECTIVE

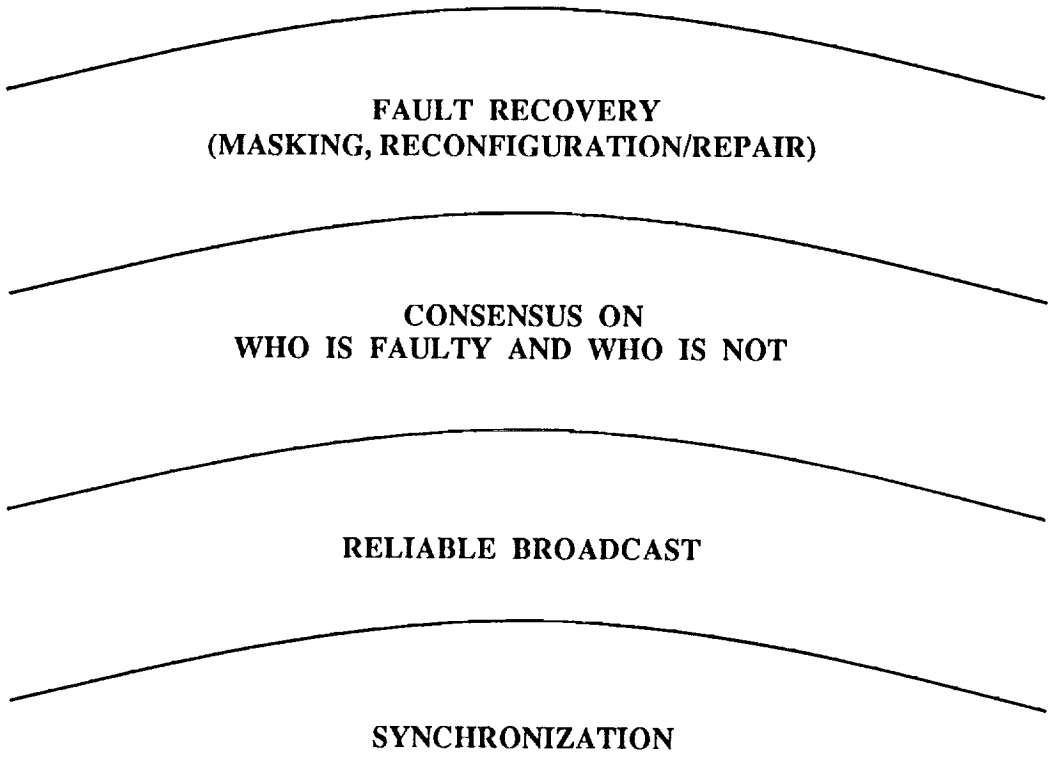


Figure 1.1: A four-layered view of fault tolerance in multiprocessor systems.

reinitiated from there and the previously faulty results are recomputed. Another form of backward recovery is achieved by the recovery block approach [13]. These techniques highly utilize time redundancy.

As can be seen from the discussed examples, fault-tolerant systems require space and/or time redundancy (see Figure 1.2(a)). Space redundancy is perceived here as additional hardware and lines of code, whereas time redundancy is usually extra time needed for fault detection, location and recovery. The tradeoff between fault masking and backward recovery can be best demonstrated by Figure 1.2(b). While fault masking lies in the portion of the curve corresponding to maximum space redundancy, backward recovery lies at the opposite extreme, namely the one corresponding to maximum time redundancy. Since fault masking is so costly in terms of system resources it is not appropriate for distributed systems at the system level.

Ideally, we would like to manage this tradeoff, namely space versus time, in an optimal way. This corresponds to the construction of systems lying in the region of the curve in Figure 1.2(b) containing point P_o. In other words, we would like to provide fault tolerance with minimum space and minimum time overheads.

In this scenario, forward recovery appears as a very attractive option because it can be achieved with very low space and time overheads. In a multicomputer system forward recovery can be implemented by using the computational results of the fault-free processors in order to recuperate the results of the faulty processors. In order for this to take place, the computations executed by different processors must be related by some known mathematical property. Another possibility is to have computations in different processors being independent of each other but, at the same time, cooperating to achieve a common goal. In any of these cases, specific application properties should be exploited for implementing fault tolerance with no hardware replication and very low time overhead in the absence of faults.

Finally, in order to be able to tolerate permanent faults with low time overhead, efficient reconfiguration schemes need to be implemented. Formally, by reconfiguration we understand a remapping of the computational graph of the application onto the processor architecture to avoid a faulty processor.

In this report we concentrate on layer four of our comprehensive approach for fault tolerance and describe in detail two of its aspects: efficient reconfiguration procedures and algorithmic fault-tolerant techniques. We present an approach which lays out theoretical foundations for efficient reconfiguration schemes. We identify a number of parameters that, depending on the system environment, identify optimization goals for developing low-overhead reconfiguration strategies. Our efforts in researching algorithm specific techniques anticipate results pointing to execution of computations reliably, with increased performance and no hardware replication.

Previous works in the area of reconfiguration are somewhat diverse and ad hoc, and there is a lack of analytical studies to evaluate the existing fault tolerance techniques and to guide future research. We have attempted to solve this difficult problem by a graph theoretic approach. In our research, we have introduced this approach and concentrated on the analysis and optimization of fault tolerance in multiprocessor systems.

Specifically, a reconfiguration model that allows a faulted job to be recovered with minimum space and time overhead and without performance degradation

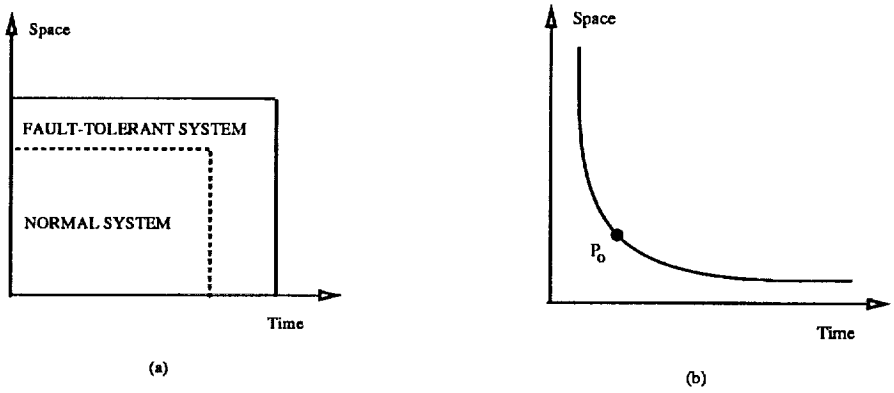


Figure 1.2: (a) Time and space redundancy needed for fault-tolerant system implementation. (b) Tradeoff between space and time redundancy required for fault tolerance.

has been formally introduced. This allows the execution of jobs on a multiprocessor system with predictable behavior. Additionally, eleven parameters have been precisely defined to facilitate the evaluation of the fault tolerance of different multiprocessor systems for executing a given set of target applications. These parameters also allow the quantitative comparison of various fault reconfiguration techniques so that efficient algorithms can be developed. The graph theoretic approach used is widely applicable to multiprocessor systems and applications with various topologies. We have concentrated on two well-known systems, namely, the mesh and the hypercube, and two frequently used computational structures, namely, the path and the complete binary tree. Solutions and algorithms for determining various optimization parameters have also been presented. Algorithms that allow optimized job reconfiguration are also developed. More importantly, we have studied the applications of the analytical approach to the fault-tolerant design of multiprocessor systems. Our approach explores the inherent fault tolerance of multiprocessor systems and exploits the topological relationship between the system architecture and the target applications. Hence our approach not only leads to good reconfiguration procedures but also helps one in designing and selecting a good architecture for fault tolerance based on the requirements of the target application.

We have proposed the novel concept of Hybrid Algorithm Technique. This approach provides a powerful means for algorithmic fault tolerance and also exhibits better performance. The idea is based upon creating a hybrid of multiple algorithms for solving a problem. The algorithms are executed simultaneously, perhaps in parallel, and the results are regularly compared. A good result is then broadcasted to all participating processes and the computation then continues from that common intermediate result. The occurrence of a fault in a particular algorithm will cause the results of that algorithm to be rejected. In addition, the new approach leads to improved performance in terms of the quality of result and/or execution time.

This research lays the theoretical foundations for the management of redundancy in NASA's distributed systems. The algorithms described in later sections will facilitate an evaluation of tradeoffs between time and space re-

dundancy for fault tolerance. A good understanding of the issues involved in these tradeoffs may help to optimize the system resources to achieve reliable computation in NASA's distributed systems.

2 THE APPROACH

Recently, Harary and Malek have developed a graph theoretic framework for fault recovery in multiprocessor systems [14][15]. In this work, existing graph theoretic models for system architecture and program structure are referred to as the architecture graph and the computation graph, respectively, and are used to formalize the studies of fault recovery. Several parameters that affect the effectiveness of a fault recovery technique in various ways are introduced to allow easier comparison of different methodologies and to quantify the optimization of fault recovery. A distinction between system synthesis and analysis are also made. The fault recovery problem is formulated as a problem of remapping the computation graph onto the architecture graph after the exclusion of faulty components. This corresponds to job reconfiguration without performance degradation. For this reason, we will use the terms fault recovery, job reconfiguration, and remapping interchangeably throughout this report.

In this section, we present a formal approach to solving the problem of multiprocessor systems fault tolerance. This approach is based on the groundwork developed in [14].

2.1 The System Model

Several graph theoretic models for multiprocessor systems have been developed. The model relevant to this research consists of the architecture graph G (system graph) and the computation graph H (program graph). The architecture graph represents the physical organization of a multiprocessor system. Nodes in this graph represent processing elements (PEs) and interface communication modules, while edges indicate the actual point-to-point communication links. Each node in this graph can be extended to depict the detailed view of the PE with its processor, memory, input/output channels, and devices. Fig. 2.1 shows the architecture graph of an 8-PE hypercube system.

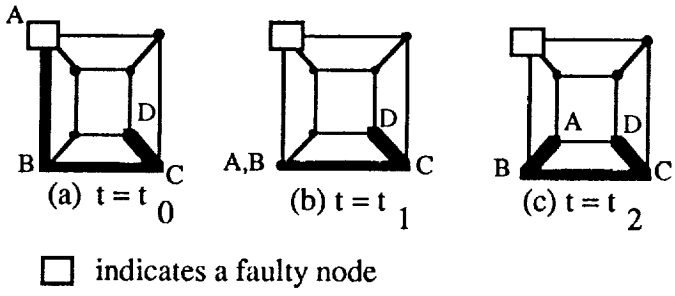


Fig. 2.1. The recovery of a P_4 on a Q_3.

2.2. The Computation Model

The computation graph represents an actual computation (job) where each node corresponds to a task and each edge indicates inter-task communications. The dark line in Fig. 2.1a shows a computation graph of a 4-node path mapped 1-1 onto an architecture graph of an 8-PE hypercube.

We recognize that a computation graph H is, in general, a digraph representing a task graph, and a 1-1 mapping of H onto G may not be possible in many cases. When H is not a subgraph of G, we can resort to the following two measures.

1. If $|H| \geq |G|$, where $|X|$ is the number of nodes in graph X, we have a problem known as cardinality variation. In this case, a graph H_R must be generated such that $|H_R| < |H|$ and H_R is a subgraph of G. Consequently, several nodes (R nodes) in H may be mapped onto a single node in H_R. It is important to keep this ratio R as small as possible.

2. If $|H| < |G|$, we have a problem known as topological variation. In this situation, a graph H_D which is homeomorphic to H due to the insertion of extra node(s) and edge(s) may be generated such that H_D is a subgraph of G. As a result, some adjacent nodes in H will be non-adjacent in H_D. The length of the longest path in H_D corresponding to an edge in H is called the dilation, D, of the embedding. It is obviously desirable to minimize D.

The generation of an H_R or an H_D with minimized R or D corresponds to the mapping problem, which is equivalent to the graph homomorphism problem. Since the general solution is NP-complete, researchers have attempted to develop good heuristics to handle this problem. In our fault recovery model, we assume that the computation structure has been mapped onto a computation graph H, which is a subgraph of G. In reality, H could correspond to H_R or H_D mentioned above.

2.3. The Fault Recovery Model

Let G be a given architecture graph and let a computation graph H be a subgraph of G. A faulty link in G leads to the removal of an edge and a faulty processor results in the removal of a node and the incident edges. When one or both of these cases occur, there are two possibilities: Either the resulting graph G' contains another subgraph H' that is isomorphic to H, or it does not. If it does not, then we call the system G non-recoverable with respect to H and the particular fault(s). Notice that non-recoverability with respect to this reconfiguration model could lead to the reduction of nodes in H, which may be handled by task redistribution. On the other hand, when G' does contain a subgraph H' isomorphic to H, and there are two or more such subgraphs, then the one yielding the minimum cost (such as some function of time, the number of new nodes required, or other parameters introduced in [14]) will result in the most efficient fault recovery. Fig. 2.1 shows the recovery of a job on a hypercube system.

According to the above fault recovery model, the general recoverability problem is a subgraph isomorphism problem, and is NP-complete. However, if H and G are graphs of regular structures such as path, ring, tree, mesh, hypercube, and so on, the problem is much more tractable. In this research we have considered G to be a mesh or a hypercube and H to be a path or a complete binary tree.

2.4. The Fault Model

Since our focus in this report is on the hypercube and the mesh, which use point-to-point communications, faults in such a system can be categorized according to their location as follows.

1. A faulty node corresponds to a fault in a PE, which results in the removal of a node from the architecture graph. This includes a fault in the CPU, memory, I/O device, data paths, and so on.

2. A faulty edge corresponds to a fault in a communication link or I/O channel, which results in the removal of an edge from the architecture graph.

Faults can also be classified according to their duration as follows.

1. Temporary faults include transient and intermittent faults [12]. This type of faults predominate in VLSI systems.

2. Permanent faults [12], which can be a result of either production defects, component wearout, or environmental factors.

Furthermore, faults can be considered as single or multiple faults. The definitions of single and multiple faults are less precise because it depends on whether a global or local viewpoint is taken. For example, faults affecting a single communication path can be classified as a single fault from the global point of view, but it may also be categorized as a multiple fault when individual links in the path are considered. In this work, we address all types of faults mentioned above. However, we would like to point out that temporary faults with short durations are usually more efficiently recovered by retry instead of reconfiguration.

2.5. The Resiliency Triple

The resiliency triple (m, r, c) consists of the following three parameters: multiplicity (m), robustness (r), and configurability (c). They are measures of the resiliency of a system G to fault(s) while running a job H. Consequently, each parameter is dependent on both the architecture graph, G, and the computation graph, H.

Definition 2.1: Multiplicity, denoted by *m(G, H)*, is the maximum number of node-disjoint embeddings of H onto G (Fig. 2.2a). In graph theory, this is known as the node-disjoint packing number $pac_O(G, H)$.

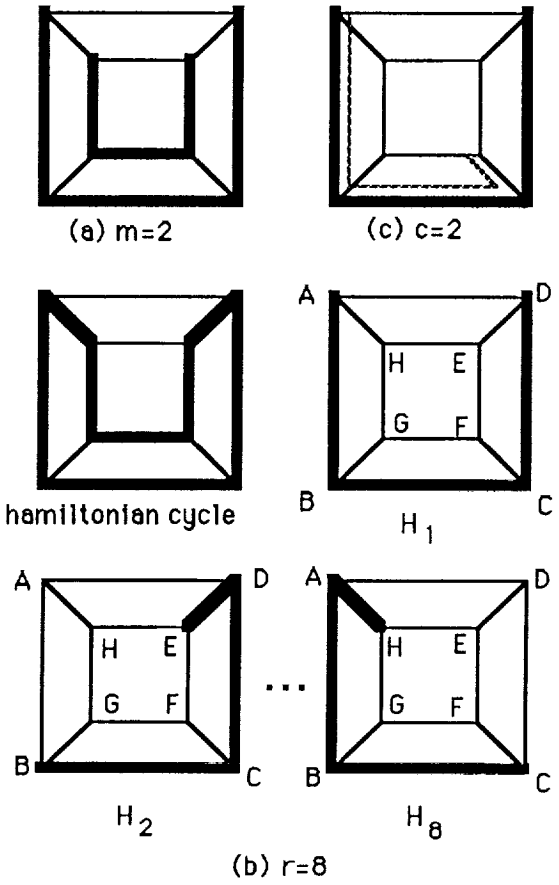


Fig. 2.2. The multiplicity, robustness, and configurability of P_4 on Q_3.

Since a 1-1 mapping of H onto G is not always possible, H could be embedded onto G by a number of mapping schemes involving bigger-than-one dilations. We use $m_e(G, H)$ to denote the multiplicity of H on G with respect to the embedding scheme, *e*. When m = 2, two identical jobs can be run simultaneously on the system (with some additional hardware) to allow single fault detection. When m > 2, any single fault can be masked by voting the outputs of multiple copies of the same job. In other words, it allows N Modular Redundancy (NMR) in space. In practice, we are only interested in knowing whether *m* is equal to or greater than a chosen number in the range of 2 to 9. Multiplicity is also an indication of a system's fault tolerance. A system with multiplicity, m, may be up to (m-1)-fault-tolerant. Higher multiplicity also allows more homogeneous jobs to be run on the system simultaneously to achieve a better system utilization. Futhermore, production testing can be performed much more efficiently than the traditional technique by comparing results of the same job executed on different subsets of processors.

Definition 2.2: Robustness, denoted by $r(G, H)$, is the number of embeddings of a graph H onto a labeled graph G such that each node of H is at a different label of G in each embedding (Fig. 2.2b).

When r > 1, fault recovery can be achieved through time redundancy by executing each stage of the computation (systolic array or pipeline) on two or more different processors at a time [16]. This corresponds to duplex or NMR in time. Again, we are usually concerned about whether *r* is equal to or bigger than a chosen number within the range of 2 to 9. Since H can be embedded onto G in various ways, we use $r_e(G, H)$ to denote the robustness of H on G with respect to a particular embedding scheme, *e*. Notice that multiplicity and robustness correspond to redundancy in space and time. They are, therefore, also useful in system diagnosis.

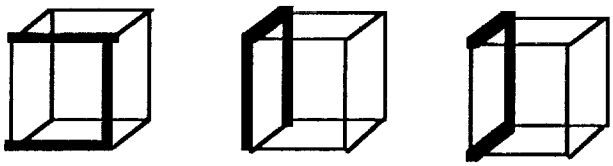
If H is a proper subgraph of G, there may be many ways to map H onto G. Each particular mapping, represented by a graph H_i (i is a positive integer), is called a configuration. Since all configurations of H on G are isomorphic, the computation at hand can be performed using any of them. However, some of these isomorphic graphs are equivalent. Although isomorphism among a collection of configurations is itself an equivalence relation, we have, for reasons that will become obvious later, defined equivalence in a stricter sense. If all configurations are considered as "rigid" graphs, then two isomorphic configurations may not possess the same properties such as dimensionality and space occupancy.

Definition 2.3: Two configurations, H_1 and H_2, are equivalent if, after some necessary rotation and/or translation, H_2 either coincides with, or becomes a mirror image of H_1.

Definition 2.4: Configurability, denoted by $c(G, H)$, is the number of non-equivalent configurations of H on G (Fig. 2.2c).

Fig. 2.3 shows some equivalent configurations of a 4-node path, P_4, on an 8-node hypercube, Q_3. As mentioned earlier, when H is not a subgraph of G, a subgraph, H_D, of G which is homeomorphic to H is embedded onto G to run the computation H. H_D is a dilation-D embedding of H onto G. Different embedding schemes may result in different H_D's. Consequently, configurability is also dependent on the embedding scheme. We use $c_e(G, H)$ to denote the configurability of H on G with respect to a particular embedding scheme, *e*. Notice that each set of equivalent configurations is counted as one in deriving the configurability of a given computation graph on an architecture graph. The parameter, configurability, is a measure of several aspects of a multiprocessor system. Higher configurability generally results in a greater multiplicity and allows a better system utilization, a greater resiliency to faults, and a higher efficiency in fault recovery.

Since each parameter in the resiliency triple has some impact on the fault tolerance of a multiprocessor system, the study of these parameters is not merely of theoretical interest, but it is also useful in solving practical problems.



(1, 2, 1) (a) (2, 3, 2) (b) (3, 2, 3) (c)

Fig. 2.3. Equivalent configurations (fixed labeling).

2.6. The Fault Recovery Vector

In order to formalize and quantify the job reconfiguration problem introduced earlier, eight other parameters, collectively called the fault recovery vector (FRV), are also introduced as the optimization criteria for fault recovery. These are defined as follows.

Definition 2.5: Distance, denoted by d = d(H, f), with respect to an isomorphism f from H to H' where f(v) = v', is equal to the sum of the distances in G of d(v, v') for all the nodes v in H. This corresponds to the sequential recovery time.

Definition 2.6: Time, denoted by t = t(G', H), is the maximum value of the terms d(v, v') in the above sum. It corresponds to the recovery time when all the data transfers are non-contending.

The distance-time pair (d, t) is called the *recovery effectiveness.*

Definition 2.7: Number of new nodes, denoted by ν_0, is the number of new nodes that are utilized in order to configure the job H' that is equivalent to H when a fault occurs in H.

Definition 2.8: Number of used nodes, denoted by μ_0, is the number of nodes in G that are traversed while moving from the faulty job H to the faultless job H', but do not appear in H U H'. These nodes are not incorporated in the resulting subgraph of the recovered job.

Definition 2.9: Number of new edges, denoted by ν_1, is the edge counterpart of ν_0. It is the number of new edges in the newly recovered job H' that are not used in the original faulty job H.

Definition 2.10: Number of used edges, denoted by μ_1, is the number of edges that are utilized while mapping H onto H'. These used edges enable the transfer of program code and data from H to H'. They do not occur in H U H'.

The quadruplet of parameters (ν_0, μ_0, ν_1, μ_1) is called the *recovery overhead.*

During a recovery procedure there are certain non-faulty nodes which can remain in their original locations in G. These are called the *stationary nodes*. A node which must be moved in order to reconfigure is called *relocated.*

Definition 2.11: Relocation order, denoted by ρ_0 is the number of transferred nodes in a reconfiguration procedure.

Definition 2.12: Relocation size, denoted by ρ_1, is the number of transferred edges during reconfiguration.

The pair (ρ_0, ρ_1) is called the *relocation measure.*

The fault recovery vector (FRV) is defined as the vector of eight parameters formed by concatenating (d,t), $(\nu_0, \mu_0, \nu_1, \mu_1)$, and (ρ_0, ρ_1).

The above problem formulation and the concise definitions of the eleven fault tolerance optimization parameters provide an excellent tool for analytical studies in the area of multiprocessor fault tolerance. Based on this groundwork, we will, in the following sections, demonstrate how this approach can be used to analyze and optimize fault tolerance in multiprocessor systems. We will also show how the choice of a system architecture G for a computation graph H, the location of a fault, and a particular reconfiguration procedure can affect the fault recovery parameters.

Next, we present the methods developed for determining the resiliency triple for a computation graph H on an architecture graph G. Since the path and the binary tree are two very frequently used computation structures, and a path can be effectively used for solving the general resource allocation problem, we have decided to consider the computation graph H to be a path or a binary tree. The mesh and the binary n-cube (hypercube) have both received much research and commercial attention, and are useful for a wide range of problems. We, therefore, choose these two systems as the architecture graphs, G's, under consideration. For clarity of presentation, we focus on complete binary trees and square meshes. However, results for arbitrary binary trees and rectangular meshes can be obtained with easy modifications.

3 THE RESILIENCY TRIPLE IN MESH AND HYPERCUBE MULTIPROCESSOR SYSTEMS

In Section 2, we have defined the resiliency triple, which includes the multiplicity, the robustness, and the configurability, collectively denoted by (m, r, c). These parameters play an important role in the better utilization of a multiprocessor system, its resiliency to faults, and its suitability for various fault recovery strategies [17]. This section presents methods developed to determine these parameters for two important computation graphs, the path and the complete binary tree, on two well-known architecture graphs, the mesh and the hypercube. Without loss of generality, we shall assume an s × s square mesh and denote it by M_s, where s is the number of nodes along the horizontal or vertical dimension. We will also use Q_n to denote an n-dimensional hypercube, P_k to denote a path that consists of k nodes, and T_l to denote an l-level complete binary tree.

3.1. The Resiliency Triple in a Mesh System

3.1.1. A Path on a Mesh

Multiplicity

If a computation graph H contains N' nodes and an architecture graph G contains N nodes, and $N' \leq N$, then by Definition 1.1, the multiplicity of H on G is at most $\lfloor N/N' \rfloor$. That is, $m(G, H) \leq \lfloor N/N' \rfloor$. Notice that $\lfloor x \rfloor$ is the largest integer
smaller than or equal to x.

Theorem 3.1: Given a computation graph P_k, which is a path containing k nodes, and an architecture graph G, which contains N nodes, if a hamiltonian path exists on G, then $m(G, P_k) = \lfloor N/k \rfloor$.

Proof: Since G contains a hamiltonian path, P_N, and $N \geq k$, node-disjoint copies of a path P_k can be concatenated along P_N until less than k nodes are left. This will allow $\lfloor N/k \rfloor$ node-disjoint copies of P_k to be mapped on G. □

Corollary 3.1: The multiplicity of a computation graph, P_k, on an architecture graph, M_s, is given by $m(M_s, P_k) = \lfloor s^2/k \rfloor$.

Proof: Since a hamiltonian path exists on a mesh of any size, and there are s^2 nodes in M_s, the proof follows from Theorem 3.1. □

Robustness

From Definition 3.2, each node in a computation graph H must be mapped onto a different labeled node of the N-node architecture graph G to obtain an eligible embedding. Since there are N labels in G, the robustness, which is the maximum number of eligible embeddings, is N. That is, $r(G, H) \leq N$.

Theorem 3.2: Given a computation graph, P_k, which is a k-node path and an architecture graph, G, which contains N nodes, if a hamiltonian cycle exists on G, then $r(G, P_k) = N$.

Proof: In the hamiltonian cycle, C_N, that exists on G, let us use the label *A* to denote an arbitrary node. Let us also use the label *a* to denote the head of the path, P_k, and the label *b* to denote the tail of P_k. We can map P_k on C_N such that *a* coincides with *A*. If we call this embedding H_1, then the next embedding, H_2, can be obtained by moving each node in H_1 to an adjacent node in the same direction along C_N. This process can be continued to obtain different mappings until node *a* in P_k returns to node *A* in C_N, which gives H_1. Since C_N contains N nodes, N different embeddings are generated, indicating that the robustness is N. □

Corollary 3.2: If the architecture graph is a mesh with an even number of nodes, M_s^e, and the computation graph is a k-node path, P_k, then the robustness is given by $r(M_s^e, P_k) = s^2$.

Proof: Since there exists a hamiltonian cycle on any mesh with an even number of nodes, and there are s^2 nodes on M_s^e, the proof follows from Theorem 3.2. □

Theorem 3.3: If the architecture graph is a mesh with an odd number of nodes, M_s^o, and the computation graph is a k-node path, P_k, then the robustness is given by $r(M_s^o, P_k) = s^2 - 1$.

Proof: Since a hamiltonian cycle does not exist on a mesh which contains an odd number of nodes, two alternatives which are easy to implement can be taken. (1) Find the largest cycle in M_s^o and use it instead of the hamiltonian cycle to generate the mappings described earlier. (2) Slide P_k along the hamiltonian path, P_N, on M_s^o ($N = s^2$), one node at a time, from one end of P_N to another. Using option (2), the robustness will be given by $r(M_s^o, P_k) = s^2 - k + 1$. Clearly, if we can find a cycle in M_s^o such that $N' > s^2 - k + 1$, where N' is the number of nodes in this cycle, then option (1) will give a better result. We observe that the largest cycle in M_s^o contains N - 1 nodes. Fig. 3.1 shows how such a cycle is constructed on an arbitrary M_s^o.

Consequently, using option (1), the robustness is given by $r(M_s^o, P_k) = s^2 - 1$. This

is optimum (for $k > 2$) because a hamiltonian cycle does not exist on an M_s^o. □

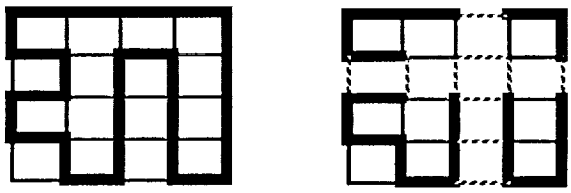


Fig. 3.1. The largest cycle on M_s when s is odd.

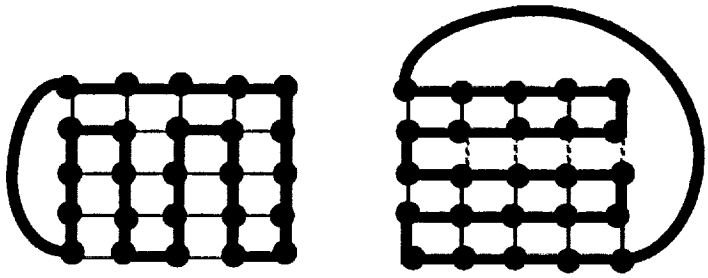
We would like to point out that for a mesh with wrapped around connections, such as a tube or a torus, a hamiltonian cycle always exists. In fact, if an extra edge is added between any two corner nodes to a mesh, M_s^o, of an arbitrary size, a hamiltonian cycle is guaranteed. If M_s' is one of the modified meshes mentioned above, then $r(M_s', P_k) = s^2$. Fig. 3.2 shows two hamiltonian cycles on a modified M_5 with an extra edge connecting different corner nodes.

Configurability

In order to determine configurability, we need to generate various non-equivalent configurations. Before presenting an algorithm to accomplish this, we shall describe a scheme which is suitable to represent a configuration of a path P_k on a mesh M_s. It is also helpful to observe the following characteristics of a square mesh, M_s.

1. The M_s has four corner nodes, which are of degree 2, and 4s - 8 boundary nodes, which are of degree 3. The remaining nodes all have a degree 4. Notice that in a torus, which is a wrapped around mesh, all nodes have a degree 4.

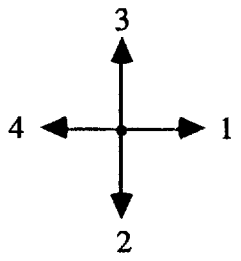
2. A configuration of P_k on M_s has three rotations besides itself. It also has two mirror images, one along the x-axis (horizontal) and the other along the y-axis (vertical).



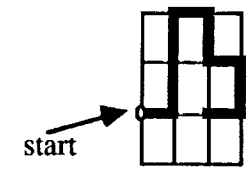
(a) (b)

Fig. 3.2. Hamiltonian cycles on M_5 with an extra edge.

We have decided to ignore the boundary cases on M_s in order to simplify the discussion. This requires that $k \leq s$. However, if $k > s$, then the number of consecutive edges traversed in each direction must be counted so as not to exceed s. Since each node under consideration has a degree of 4 regardless of s, and given an edge in a path, the next edge to be traversed can only be oriented in one of three directions (two adjacent edges in the path cannot overlap on a mesh), we have chosen to assign a fixed integer to each of the four directions, as shown in Fig. 3.3a. Since P_k has k-1 edges, a configuration of P_k on M_s can be represented by a (k - 1)-integer vector as shown in Fig. 3.3b.

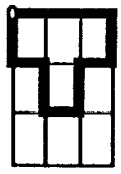


(a) labeling each direction



v = (1, 3, 3, 1, 2, 1, 2, 4)

(b) a path



v = (1, 1, 1, 2, 4, 2, 4, 3, 4, 3)

(c) a cycle

Fig. 3.3. Representation of a path and a cycle on a mesh.

We may require that the first integer be 1 (first edge always heads to the right) for easier reference. When k = 2, there is only one configuration, which is represented by (1). When k > 2, we need to find an additional k - 2 integers to complete the (k - 1)-edge path. Because each integer can assume one of three values as mentioned earlier, up to 3^{k-2} vectors may be generated. These include many cycle-bearing or equivalent configurations. For a typical case of k = 10, $3^8 = 6561$ operations are required. Although this may be acceptable, we can improve the time efficiency in the following fashion. A configuration for P_k can be obtained from that of P_{k-1} and ultimately from P_2 by appending an edge to either end of the latter, step by step until k-1 edges are accumulated. To obtain P_i from P_{i-1}, $6c(M_s, P_{i-1})$ vectors are generated. Thus, the number of operations required for the complete process is given by

$$\sum_{i=3} 6c(M_s, P_{i-1})$$

If we put an upper bound, x, on the number of non-equivalent configurations generated for each P_i, where $2 < i \leq k$, then O(k) computing time is needed. This reduces to O(1) if we further assume that K is an upper bound on the number of nodes in P_k.

After generating the configuration vectors mentioned above, we need to perform the eligibility test. This consists of cycle detection and equivalence test. The following theorem can be used for cycle detection.

Theorem 3.4: Given the labeling scheme in Fig. 3.3a and a vector v representing a configuration on a mesh M_s, if we let *sum* denote the sum of all the integers in v and *length* denote the number of integers in v, then the configuration is a cycle *iff* ***sum* = 2.5 × *length*.**

Proof: If a cycle on a mesh is traversed starting from an arbitrary node, each edge in the cycle would belong to a pair of edges pointing in opposite directions. A cycle of *length* edges consists of *length*/2 such pairs. Since in the introduced labeling scheme (Fig. 3.3a) each direction is numbered in such a way that integers representing opposite directions add to 5, each pair of these edges are denoted by integers whose sum is 5. Since there are *length*/2 pairs, the sum of all the integers, each representing an edge in the cycle, is 5 × *length*/2, or **2.5 × *length*.** □

Fig. 3.3c shows an example in which v = (1,1,1,2,4,2,4,3,4,3). From this we get *sum* = 1+1+1+2+4+2+4+3+4+3 = 25, *length* = 10. Applying Theorem 3.4, a cycle is detected. A vector corresponding to a cycle-bearing configuration would have a subvector that demonstrates the above characteristic. Notice that if we had chosen to label the horizontal directions (East and West) with +1 and -1 and the vertical directions (North and South) with +2 and -2, the following would have been true: ***sum* = 0**. However, we consider it rather cumbersome to use vectors of

signed integers.

We have made the following observations on equivalent configurations of P_k on M_s: Consider the example shown in Fig. 3.4a. If the path is traced starting from node A, then the following vector is obtained: H_1 = (1, 2, 2, 4, 4, 3, 1). But if node B is the starting point, then we would get H_1' = (4, 2, 1, 1, 3, 3, 4). We know that H_1 and H_1' are equivalent, but how do we detect the equivalence?

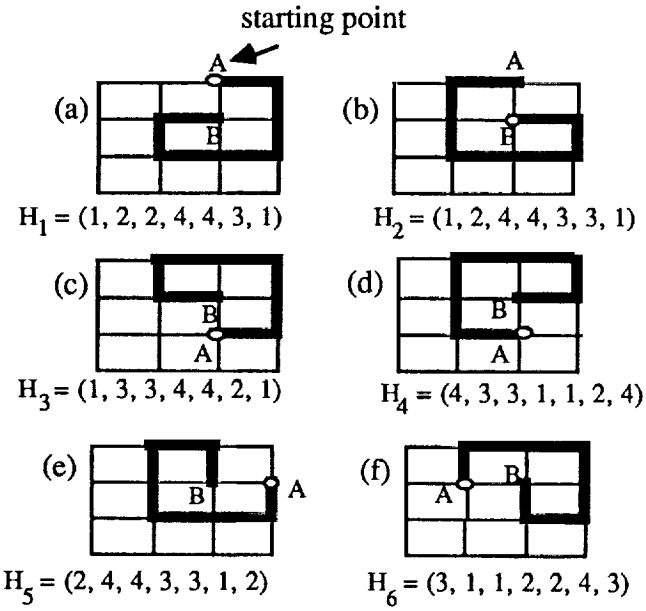


Fig. 3.4. Mirror images and rotations of a path on a mesh.

Since we have chosen 1 to be the first integer in all vectors, H_1' needs to be renumbered. In order to convert 4 to 1, we realize that the edges heading left must be forced to head right. Since mirror images are equivalent, we can convert H_1' to its mirror image along the y-axis, causing the horizontal edges to exchange directions. As a result, the integers 1 and 4 are interchanged, giving the vector $H_1' \equiv H_2$ = (1, 2, 4, 4, 3, 3, 1). Fig. 3.4b shows the corresponding configuration. Similarly, the mirror image of a configuration along the x-axis causes the vertical edges to exchange directions, resulting in the interchanging of 2 and 3 in the corresponding vector. Fig 3.4c shows such a mirror image (of the path in Fig.3.4a). Fig. 3.4d shows the path in 3.4a reflected twice, once along the x-axis and once along the y-axis. The corresponding vector is obtained by interchanging 1 and 4 as well as 2 and 3. This is equivalent to subtracting each integer in the original vector from 5. Clearly, interchanging 1 and 4 or 2 and 3 in a vector does not result in a new (non-equivalent) configuration. Finally, let us observe the configurations in Figs. 3.4e and 3.4f. These are both 90 rotations of Fig. 3.4a, one clockwise and the other counterclockwise. When a configuration is rotated 90 clockwise, integers in the

original vector must be renumbered according to the following:

$$1 \rightarrow 2, \quad 2 \rightarrow 4, \quad 3 \rightarrow 1, \quad 4 \rightarrow 3 \qquad \text{(a)}$$

When a configuration is rotated 90 ò counterclockwise, the vector must be renumbered as follows:

$$1 \rightarrow 3, \quad 2 \rightarrow 1, \quad 3 \rightarrow 4, \quad 4 \rightarrow 2 \qquad \text{(b)}$$

Thus, renumbering a vector according to (a) or (b) would not alter the configuration (all resulting configurations are equivalent). After observing the above, it is readily seen that whenever we have a vector whose first element (i) is not 1, the vector can be renumbered (to begin with 1) as follows:

1. If i=2, then reassign integers according to (b).
2. If i=3, then reassign integers according to (a).
3. If i=4, then interchange 1 and 4.

As mentioned earlier, two adjacent edges cannot be in opposite directions. Therefore, when appending an edge to the front of an existing path (represented by a vector that begins with 1), only three choices are available. The edge may be represented by one of the following three integers: 1, 2, or 3. Whenever a vector is inverted or extended at the front, renumbering may be required. A configuration (H_1 in Fig. 3.4a) and its mirror image (H_3 in Fig. 3.4c) along the x-axis both have vectors that begin with 1. It is therefore necessary to check for these equivalent configurations.

Now we are ready to present the algorithm for enumerating up to x (a chosen constant) non-equivalent configurations of P_k on M_s. The final configuration vectors are stored in an x by (k-1) array, **H**(1:x, 1:k-1), which contains up to x vectors of k - 1 integers, each of which represents a unique configuration. Then **H**(*m*, 1:k-1) would correspond to the (k - 1)-integer vector representing the *m*-th configuration enumerated. An array **T**(1:x, 1:k-1) is used to save the intermediary vectors (for P_{i-1}'s). The algorithm is as follows:

Algorithm 3.1:
Input: s, k, x.
1. If k=2, **exit** with **H**(1)=1, c=1.
2. If k>2, set **H**(1, 1)=1.
3. Until **H** contains vectors of k-1 elements, set **T**=**H**, erase **H**, and do the following:
A. For every vector **v** in **T**, **do** the following:
a. For every integer i such that $1 \le i \le 4$ and $i + j \ne 5$, **do** the following:
\j is the last integer in **v**\
i. Append i to the end of **v**.
ii. Perform cycle detection. If positive, go to 3a.
iii. Invert and renumber the vector; check if it exists in **H**. If positive, go to 3a.
iv. Interchange 2 and 3 in the vector; check if the resulting vector exists in **H**. If positive, go to 3a.
v. Invert and renumber the vector; check if it exists in **H**. If positive, go to 3a.
vi. Save the vector in **H**. If |**H**| = x, go to 3; otherwise, go to 3a.
b. For every integer i such that $1 \le i \le 3$, **do** the following:

i. Append i to the front of v and renumber the resulting vector. Check if it exists in **H**. If positive, go to 3b.
ii. Perform cycle detection. If positive, go to 3b.
iii. Invert and renumber the vector; check if it exists in **H**. If positive, go to 3b.
iv. Interchange 2 and 3 in the vector; check if the resulting vector exists in **H**. If positive, go to 3b.
v. Invert and renumber the vector; check if it exists in **H**. If positive, go to 3b.
vi. Save the vector in **H**. If |**H**| = x, go to 3; otherwise, go to 3b.
4. If |**H**| < x, set c = |**H**|, **output**("c is equal to", c). Otherwise, **output**("c is at least", x).

In Algorithm 3.1, Step 3 is repeated k-2 times. Each iteration of Step 3 causes Step 3A to be executed up to x times, each of which in turn performs Steps 3a and 3b three times. Steps 3a and 3b each requires $O(k^2)$ computing time. As a result, the total time requirement for Algorithm 3.1 is $O(k^3)$. If $k \leq K$, where K is a constant upper bound on k, this reduces to O(1). Since two x by (k-1) arrays are used for storage, the memory requirement is O(k), or O(1) if k is bounded.

3.1.2. A Complete Binary Tree on a Mesh

There is a fundamental difference between mapping a complete binary tree (T_l) on a mesh (M_s) and mapping a path (P_k) on a mesh. Given a P_k and an M_s, if $k \leq s^2$, then P_k is a subgraph of M_s. Consequently, a 1-1 mapping of P_k onto M_s is possible. However, given a T_l and an M_s, where $2^l - 1 \leq s^2$, T_l is not a subgraph of M_s except for $l \leq 4$. This means that in many realistic situations ($l > 4$), T_l cannot be mapped 1-1 onto M_s. A subgraph of M_s, T_l^D, which is homeomorphic to T_l must be used to emulate T_l. Different T_l^D's may result from different embedding schemes. Each particular T_l^D is an embedding of T_l onto M_s. The corresponding embedding function, e: $T_l \rightarrow T_l^D$, maps each node, i, in T_l onto a different node, j, in T_l^D and each edge, (i, j), in T_l onto a unique path, (f(i), f(j)), in T_l^D.

Definition 3.1: If we let E denote the edge set of T_l, d(i, j) denote the distance between nodes i and j, and D denote the dilation of the mapping, then $D = \max(d(f(i), f(j)), \forall (i, j) \in E)$.

In the previous section, we have seen that D = 1 for mapping P_k onto M_s. This is an ideal case in which the computation graph H is a subgraph of the architecture graph G. In many situations, H is not a subgraph of G and an efficient embedding scheme must be developed to map H onto G to minimize the number of wasted nodes and the extra delays introduced. The dilation, D, is a fair indication of both these items to be minimized. Therefore, a mapping scheme resulting in small D is desirable.

The complete binary tree, T_l, is relatively difficult to be mapped efficiently onto a square mesh, M_s. Several researchers have proposed area-efficient embeddings of T_l

on M_s [18][19][20]. Although Youn and Singh have proposed an embedding scheme [18] that results in higher area efficiency and smaller propagation delay than the classical H-tree scheme, they require that some nodes be diagonally connected so that the underlying system is no longer a mesh. Gordon proposed an even more efficient embedding scheme that requires some PEs (nodes in G) to act as both tree nodes and connector nodes [19]. This results in overlapped mapping. A popular embedding scheme that maps a complete binary tree, T_l, onto a square mesh, M_s, is the well-known H-tree approach [20][21]. Fig. 3.5 shows an H-tree embedding of a T_7 onto an M_{15}. From Fig. 3.5, we observe that if a T_l is mapped on an $s_1 \times s_2$ mesh, then

$$s_1 = s_2 = 2^{(l+1)/2} - 1 \qquad \text{if } l \text{ is odd} \qquad (1)$$

$$s_1 = 2^{(l+2)/2} - 1 \quad \text{and} \quad s_2 = 2^{l/2} - 1 \qquad \text{if } l \text{ is even} \qquad (2)$$

Furthermore,

$$D = 2^{(l-3)/2} \qquad \text{if } l \text{ is odd}$$

$$D = 2^{(l-2)/2} \qquad \text{if } l \text{ is even.}$$

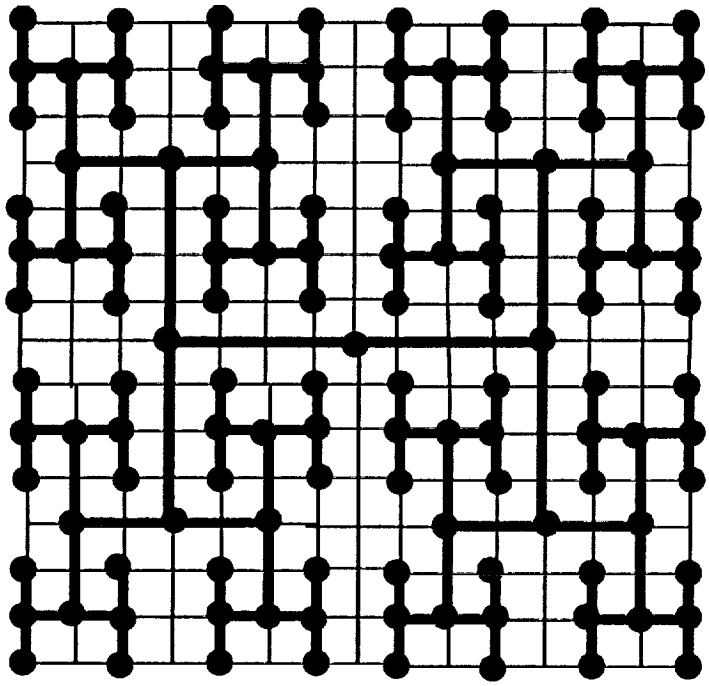


Fig. 3.5. An H-tree embedding of a T_7 on an M_{15}.

Multiplicity

According to the H-tree embedding scheme, e, in order to embed a T_l onto a given M_s, the following conditions must be satisfied: $s_1 \leq s$ and $s_2 \leq s$. Then the multiplicity with respect to e is the number of node-disjoint $s_1 \times s_2$ submeshes that exist on M_s. This is given by

$$m_e(M_s, T_l) = \lfloor s/s_1 \rfloor \times \lfloor s/s_2 \rfloor$$

From equations (1) and (2),

$$m_e(M_s, T_l) = \lfloor s/(2^{(l+1)/2}-1) \rfloor^2 \qquad \text{if } l \text{ is odd}$$

$$m_e(M_s, T_l) = \lfloor s/(2^{(l+2)/2}-1) \rfloor \times \lfloor s/(2^{l/2}-1) \rfloor \qquad \text{if } l \text{ is even.}$$

Robustness

To generate embeddings such that each node in a T_l resides on a different node in M_s, we can start by placing the $s_1 \times s_2$ submesh of M_s, which contains T_l, at the upper left corner of M_s and move it to the right one node-position at a time until we reach the upper right corner. The number of embeddings thus generated is $s - s_2 + 1$. We then move the submesh down one node-position and start sliding to the left one node at a time. We continue this process until both lower corners of M_s have been visited. The number of node-positions the submesh has moved down is $s - s_1 + 1$. Therefore, the total number of embeddings is $(s - s_1 + 1)(s - s_2 + 1)$. From equations (1) and (2), the robustness of a T_l on an M_s with respect to e is given by

$$r_e(M_s, T_l) = (s - 2^{(l+1)/2} + 2)^2 \qquad \text{if } l \text{ is odd}$$

$$r_e(M_s, T_l) = (s - 2^{(l+2)/2} + 2)(s - 2^{l/2} + 2) \qquad \text{if } l \text{ is even.}$$

Notice that in an Illiac IV-type system, M_s, which is a mesh system with wrapped around connections (a torus), $r_e(M_s, T_l) = s^2$. This is because there is no boundaries on the torus.

Configurability

With respect to the H-tree embedding scheme, e, the configurability of T_l on M_s is one because each node in T_l must maintain a constant relative position with respect to every other node. Thus, $c_e(M_s, T_l) = 1$.

3.2. The Resiliency Triple in a Hypercube System

3.2.1. A Path on a Hypercube

In an n-dimensional hypercube, Q_n, there are $N = 2^n$ nodes. Q_n is a regular graph in which every node has the same degree, n. A configuration of P_k in any particular position on Q_n can be rotated n - 1 times before returning to its original position.

Multiplicity

Corollary 3.3: The multiplicity of a computation graph, P_k, on an architecture graph, Q_n, is given by $m(Q_n, P_k) = \lfloor 2^n/k \rfloor$.

Proof: Since a hamiltonian path exists on a hypercube of any size, and there are 2^n nodes in Q_n, the proof follows from Theorem 3.1. □

Robustness

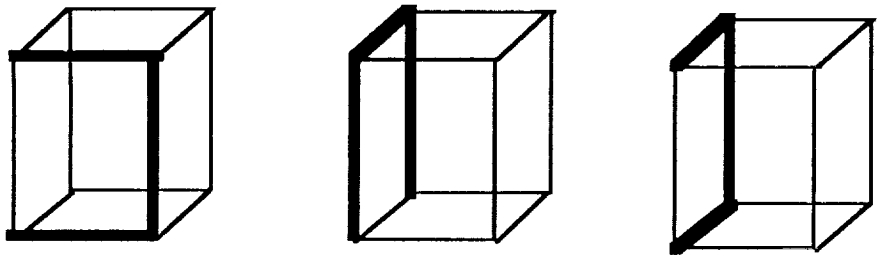
Corollary 3.4: The robustness of a computation graph, P_k, on an architecture graph, Q_n, is given by $r(Q_n, P_k) = 2^n$.

Proof: Since there exists a hamiltonian cycle on any hypercube, Q_n ($n > 1$), and there are 2^n nodes in Q_n, the proof follows from Theorem 3.2. □

Configurability

Similar to the configuration vector described in Section 3.1.1, we choose to represent a path P_k on a hypercube Q_n by a vector of k - 1 positive integers, each of which indicates the dimension in which the corresponding edge resides. Since non-equivalent configurations are not distinguished by their positions or orientations in the system, and the hypercube is a symmetric graph, we need not adopt a fixed coordinate system. Furthermore, it is more advantageous not to assign a fixed integer to each dimension. This can be demonstrated by the following example.

Example 3.4: In a Q_3 system, if the dimensions are labeled such that the horizontal dimension (x-coordinate) is denoted by 1, the vertical dimension (y-coordinate) is denoted by 2, and the remaining dimension (z-coordinate) is denoted by 3, then the configuration in Fig. 3.6a will be represented by the vector (1, 2, 1), and those shown in Figs. 3.6b and 3.6c will be represented by the vectors (2, 3, 2) and (3, 2, 3) respectively (for easier reference, we have chosen the lower left node as the starting point of the path). Obviously, all three configurations are equivalent and should be counted as one. But many such equivalent configurations will be generated as different vectors if each dimension is assigned a fixed integer. However, if we label every dimension dynamically, according to the order in which they are traversed by the path, then all the above three configurations will be represented by the vector (1, 2, 1). Fig. 3.7 shows all the non-equivalent configurations of P_6 on Q_3 using this representation. Since there are four non-equivalent configurations, the configurability of P_6 on Q_3, $c(Q_3, P_6)$, is equal



(a) (1, 2, 1) (b) (2, 3, 2) (c) (3, 2, 3)

Fig. 3.6. Path representation using fixed labeling.

(a) (1, 2, 1, 3, 2) (b) (1, 2, 1, 3, 1)

(c) (1, 2, 3, 2, 1) (d) (1, 2, 3, 1, 2)

Fig. 3.7. Path representation using dynamic labeling.

to four.

The configurability of P_k on Q_n can be obtained by enumerating all the non-equivalent configurations of P_k on Q_n. But since the distinct configurations themselves are also very useful for task allocation and fault recovery, we want to generate and save the vectors which represent them. When there are many non-equivalent configurations of H on G, we may decide to save only a chosen number, say x, of them in order to save time and memory space. In this case, we are only interested in knowing the exact number when configurability is smaller than x.

Before presenting the algorithm for generating non-equivalent configurations of P_k on Q_n, let us discuss some related issues. Firstly, we observe that when $k = 2$ or $k = 3$, there is only one configuration, represented by the vectors (1) and (1, 2)

respectively. As a result, to find a vector that represents a configuration of P_k (k - 1 edges) on Q_n, where k > 3, we only need to generate k - 3 integers to be appended to the vector (1, 2). Secondly, for any path mapped on a hypercube, no two adjacent edges can lie in the same dimension. This means that in the vector representing a configuration of P_k on Q_n, adjacent integers cannot be equal. Consequently, given an edge represented by an integer i, the next edge in the path can only be represented by an integer j such that $1 \leq j \leq n$ and $j \neq i$. In other words, we can only choose from n - 1 integers to represent this edge. Having observed this, it is clear that we may generate up to $(n - 1)^{k-3}$ vectors to be candidates for the configurations of P_k on Q_n. Many of these vectors represent configurations that contain cycles and must thus be eliminated. Other vectors may represent equivalent configurations and must thus be counted as one. Fig. 3.8 shows two configurations with cycles.

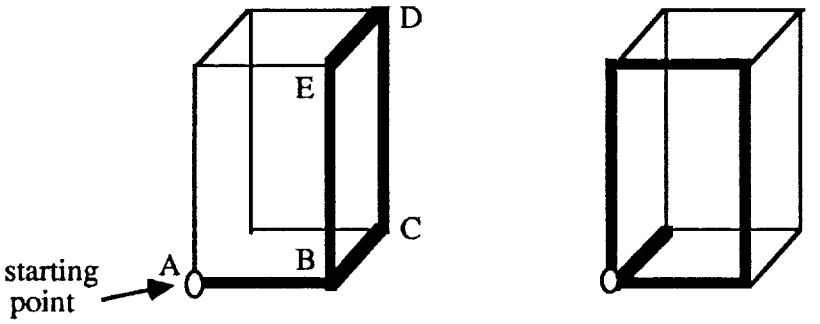


Fig. 3.8. Configurations with cycles on a Q_3.

Observably, these configurations are also equivalent. If we have to test each of the $(n - 1)^{k-3}$ vectors for cycles and equivalence, the $O((\log N)^{k-3})$ computing time may be excessive ($N = 2^n$ is the number of processors in Q_n). For a typical case of n = 10 and k = 10, the number of operations becomes $9^7 = 4{,}782{,}969$. However, if we take another approach by building the configurations of P_k from those of P_{k-1} on Q_n, then only $2(n-1)c(Q_n, P_{k-1})$ vectors will be generated. This is because we can build a path of k nodes by appending a node either to the front or to the end of a (k-1)-node path. Knowing that the configuration of P_3 is (1, 2), we can extend the path one edge at a time until we reach P_k. As a result, the number of vectors that need to be generated to obtain all the configurations of P_k on Q_n is given by

$$\sum_{i=4}^{k} 2(n-1)c(Q_n, P_{i-1}) .$$

If we put an upper bound, x, on the $c(Q_n, P_{i-1})$'s, O(klogN) computation time is

required for the whole operation. The latter approach is also more efficient in terms of memory requirement. O(k) memory space is required instead of $O((\log N)^{k-3})$, which is necessary for the former approach.

After generating the configuration vectors mentioned above, we need to perform the eligibility test. This consists of testing for the existence of cycles and equivalent configurations. In order to detect cycles, let us observe that any cycle on Q_n is represented by a vector in which every integer appears for an even number of times. Any vector that corresponds to a configuration which contains a cycle must have a subvector that exhibits the above characteristic. Let us scan a vector from the left to the right and keep an n-bit binary number as a parity indicator, in which the i-th bit, b_i, indicates whether the integer i has appeared for an even number of times. If it has, b_i is set to 0; otherwise b_i is set to 1. For example, if we consider the vector (1, 2, 1, 2, 3) in Fig. 3.8b, the 3-bit parity indicator ($P = b_1b_2b_3$) will be updated as follows when we scan the vector from the left to the right one bit at a time:

step 1:	100	(1 appeared once)
step 2:	110	(2 appeared once)
step 3:	010	(1 appeared twice)
step 4:	000	(2 appeared twice)

After scanning the fourth integer in the vector, the parity indicator becomes zero, indicating the detection of a cycle. Notice that if the first edge in the configuration is part of a cycle, as shown in Fig. 3.8b, then the parity indicator would become zero as soon as a cycle is detected, even though there is still another edge attached to the cycle. In this case, there is no need to scan the rest of the vector. However, if the first edge is not part of the cycle, as shown in Fig. 3.8a, then the parity indicator would not go to zero if the vector is scanned as a whole. In this case, the cycle will be detected when the subvector (2, 3, 2, 3) is scanned. Thus, cycle detection requires scanning the configuration vector and its subvectors while updating and checking the parity indicator. This requires $O(k^2)$ computations for P_k (O(k) if all the subvectors are checked in parallel).

The way to test for equivalence is by observing that any configuration can be traced from either end. Consider the configuration in Fig. 3.8a. If the configuration is traced in the order ABCDEB, , then we get the vector (1, 2, 3, 2, 3). But if we traverse in the opposite direction starting with node B, then the vector obtained would be (1, 2, 1, 2, 3). How do we derive this vector from (1, 2, 3, 2, 3) and thus detect the equivalence? The answer is by inverting the vector (i.e. listing a given vector by starting with the last element) and renumbering the resulting vector as follows.

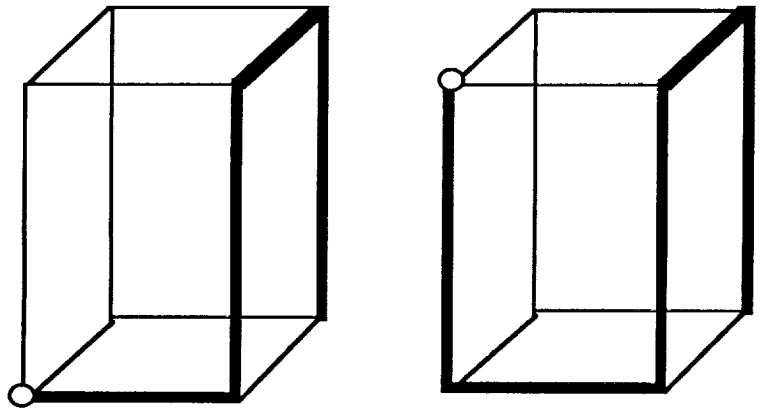
$$H_1 = (1, 2, 3, 2, 3)$$
$$H_1' = \text{INVERT}(H_1) = (3, 2, 3, 2, 1)$$
$$H_2 = \text{RENUMBER}(H_1') = (1, 2, 1, 2, 3)$$
$$\therefore H_1 \equiv H_2$$

Observably, the operation RENUMBER performs the following mapping:

$$3 \rightarrow 1, \quad 2 \rightarrow 2, \quad 1 \rightarrow 3.$$

However, it may perform a different mapping in a different situation. Its job is to scan the current vector and relabel each integer according to the order in which it appears in the vector. In H_1', the integer 3 is the first label to appear in the vector, and is thus given a new label 1. Similarly, the integer 2 is the second label and 1 the third appearing in H_1', they are therefore reassigned the new labels 2 and 3, respectively. Since we have not assigned a fixed number to any particular dimension in Q_n, inverting and then renumbering a vector would not produce a new configuration, and is equivalent to tracing the path from the opposite end. Renumbering is also necessary when a configuration vector for P_k is generated by appending an interger to the front of a vector of P_{k-1}. For example, if we want to obtain a configuration for P_6 by appending an edge to the front of the P_5 shown in Fig. 3.9a, we can append an integer (either 2 or 3 in this case) to the front of the vector that represents the P_5. If we choose to append a 2, the following vector is obtained: $H'(P_6) = (2, 1, 2, 3, 2)$. The corresponding configuration is shown in Fig. 3.9b. $H'(P_6)$ needs to be renumberred as follows:

$$2 \rightarrow 1, \quad 1 \rightarrow 2, \quad 3 \rightarrow 3,$$
$$H(P_6) = \text{RENUMBER}(H'(P_6)) = (1, 2, 1, 3, 1).$$



(a) $H(P_5) = (1, 2, 3, 2)$ (b) $H(P_6) = (1, 2, 1, 3, 1)$

Fig. 3.9. Extension of a 5-node path to a 6-node path.

Having discussed the various issues involved in generating the configurations of P_k on Q_n, we are now ready to present the algorithm which enumerates up to x non-equivalent configurations of P_k on Q_n. As in Algorithm 3.1, two arrays, H(1:x,

1:k-1) and **T**(1:x, 1:k-1), are used to store the final and the intermediary vectors, respectively. The algorithm is as follows.

Algorithm 3.2:
Input: n, k, x.
1. If k = 2, **exit** with **H**(1) = (1), c = 1.
2. If k = 3, **exit** with **H**(1) = (1, 2), c = 1.
3. If k > 3, set **H**(1,1) = 1 and **H**(1,2) = 2; h = 2.
\\ h keeps track of the largest integer in the vector and h ≤ n \\
4. Until **H** contains vectors of k-1 elements, set **T** = **H**, erase **H**, and do the following:
A. For every vector **v** in **T**, **do** the following:
a. If h ≠ n, set h = h + 1.
b. For every integer i such that 1 ≤ i ≤ h and i ≠ j, **do** the following:
\\ j is the last integer in **v** \\
i. Append i to the end of **v**.
ii. Perform cycle detection. If positive, go to 4b.
iii. Invert and renumber the vector; check if the resulting vector has been saved in **H**. If positive, go to 4b.
iv. Save the resulting vector in **H**. **If** |**H**| = x, go to 4; otherwise go to 4b.
\\ |**H**| is the number of vectors in **H** \\
c. For every integer i such that 2 ≤ i ≤ h, **do** the following:
i. Append i to the front of **v** and renumber the vector.
ii. Check if vector exists in **H**. If positive, go to 4c.
iii. Perform cycle detection. If positive, go to 4c.
iv. Invert and renumber the vector; check if it exists in **H**. If positive, go to 4c.
v. Save the resulting vector in **H**. If |**H**| = x, go to 4; otherwise go to 4c.
5. If |**H**| < x, set c = |**H**| and **output**("c is equal to", c); otherwise, **output**("c is at least", x).

In the above algorithm, Step 4 is executed k - 3 times. For each iteration of Step 4, Step 4A is invoked at most x (a constant) times, each of which causes Steps 4b and 4c to be performed h times. Steps 4b and 4c perform cycle detection and vector renumbering, which require $O(k^2)$ computing time. Consequently, the total time requirement for Algorithm 3.2 is $O(k^3h)$. By observing that h is O(k) if k ≤ n + 1 and is O(n) if k ≥ n + 1, we conclude that the computation time is either $O(k^4)$ or $O(k^3 \log N)$. If we assume an upper bound on the size of the path so that k ≤ K, where K is a chosen constant, then the computation can be accomplished either in a constant (O(1)) time or O(logN) time, depending on the values of k and n. Since two x by (k-1) arrays are used to store the final and the intermediary results, the memory requirement for Algorithm 3.2 is O(k). Again, for k ≤ K, this means O(1) storage space.

3.2.2. A Complete Binary Tree on a Hypercube

A complete binary tree T_l can be more efficiently mapped onto a hypercube Q_n than onto a mesh M_s. A T_l is a subgraph of Q_n if $l \leq n - 1$, which allows T_l to be

mapped 1-1 onto Q_n. We assume that $|H| \leq |G|$, where $|G|$ denotes the number of nodes in G and $|H|$ denotes the number of nodes in H. Since $|T_l| = 2^l - 1$ and $|Q_n| = 2^n$, $l \leq n$. Therefore, the only case when mapping a T_l on a Q_n involves a bigger-than-one dilation is when $l = n$. Deshpande and Jenevein have shown that a T_n is optimally mappable onto a Q_n with dilation 2 [22]. By inserting a connector node between the root node of a T_n and one of its two sons, the root is connected to one subtree (T_{n-1}) by an edge and the other subtree by a path of length 2. The resulting graph is a subgraph of Q_n and can be mapped 1-1 onto the latter. Fig. 3.10 shows a T_3 mapped onto a Q_3 in this fashion.

Multiplicity

We have seen from above that using the embedding scheme, *f*, proposed in [22], a T_l can always be mapped onto a Q_n. if $l \leq n$. Thus, the multiplicity with respect to *f* is given by $m_f(Q_n, T_l) = 2^{n-l}$. This is the number of *l*-dimensional subcubes in Q_n.

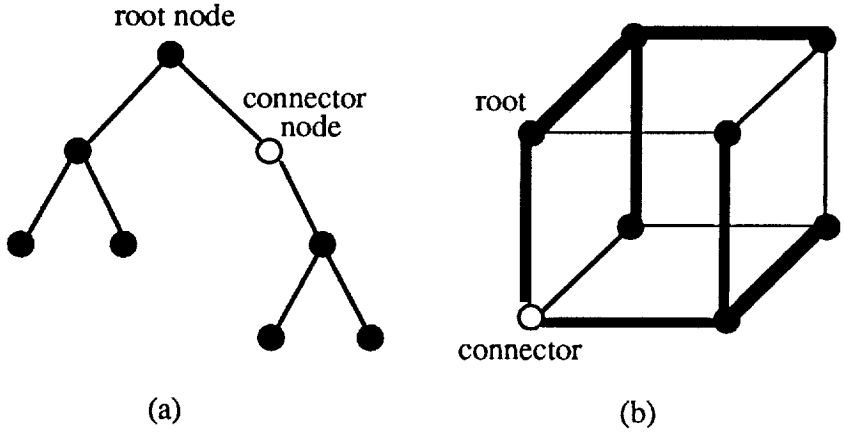


Fig. 3.10. Mapping of a T_3 onto a Q_3 with dilation 2.

Robustness

The robustness of a T_l on a Q_n is given by the following.

Theorem 3.5: $r(Q_n, T_l) = 2^n$ with respect to both the embedding scheme *f* for $l = n$ and the 1-1 mapping for $l < n$.

Proof: Let us assign an n-bit binary label to each node in Q_n such that the labels of adjacent nodes differ in only one bit position. Let us also denote an n-bit binary mask by **b**. Then if the label of each node in Q_n is EXCLUSIVE-ORed with **b**, a new label is generated for each node in such a way that the original adjacency relations among the nodes are maintained. Since **b** can assume 2^n binary values,

each node in Q_n can be relabeled 2^n - 1 times (**b** = 0...0 does not result in a new label). If we associate each node with a particular label, then relabeling Q_n corresponds to repositioning the nodes in it. Consider mapping a T_l onto a Q_n ($l \leq n$) and keeping nodes in T_l stationary. If Q_n is repositioned (relabeled) 2^n - 1 times in the manner described above, then every node in T_l will be given a new label each time. This is equivalent to remapping T_l on Q_n in such a fashion that each node in T_l has been mapped onto every label in Q_n after 2^n mappings. □

Notice that if the relabeling is performed in an order such that each label is EXCLUSIVE-ORed with a mask **b** while **b** sequentially assumes the values of 2^n labels that form a hamiltonian cycle in Q_n, then every node in Q_n will traverse along a hamiltonian cycle.

Configurability

By observing the embedding scheme, *f*, described above, we see that the configurability of a T_l on a Q_n with respect to *f* is one, or $c_f(Q_n, T_l) = 1$.

3.3 Summary of results about the resiliency triple

Methods for determining the resiliency triple for a path or a binary tree computation graph on a mesh or a hypercube system were presented. The resiliency triple for a complete binary tree on a square mesh and a hypercube were obtained with respect to two popular embedding schemes [21] [22]. For a path on these two systems, we provided the solutions for determining the first two parameters, multiplicity and robustness. We also presented two algorithms which determine the configurability and enumerate various non-equivalent configurations. The configurations generated by these algorithms are useful for efficient task allocation as well as effective fault recovery. The efficiencies of the two algorithms have been improved by carefully choosing a path representation scheme in each case, and by selecting an effective approach to generating the configurations. For bounded k, Algorithm 3.1 requires constant time and memory, while Algorithm 3.2 has O(logN) computing time and requires O(1) storage space.

3.4 Optimization of the fault recovery vector

In Section 2, we have defined the fault recovery vector, FRV = (d, t, ν_0, μ_0, ν_1, μ_1, ρ_0, ρ_1). The FRV contains eight important parameters by which the fault recovery efficiency can be measured. It allows quantitative comparisons of various fault recovery alternatives. Here we address the optimization of fault recovery with respect to a selected subset of the FRV, namely, recovery time t and node overhead ν_0.

The fundamental issue is how to effectively remap a given computational graph, which has been mapped into an architecture, when a fault occurs. Given a computational graph that is embedded onto an architecture, if one of its nodes,

representing a processor, is plagued by a fault, the process of reconfiguration consists of finding a new embedding of the same computational graph onto the modified architecture. The modification in the architecture is basically the unavailability of the faulty node. Another consideration is that a given computational graph can be embedded in different ways onto a given architecture. We call a *configuration* each one of these different embeddings.

We have developed algorithms for remapping different computational graphs onto different architectures, using different initial configurations. Namely, we have worked with two widely used computational graphs, the path and the complete binary tree, and two popular multiprocessor architectures, the mesh and the hypercube. For different configurations, the algorithms show that different space and time overheads are necessary for achieving fault tolerance. These results give the user the freedom of choosing the configuration that best represents his priorities in terms of extra nodes (space) and extra time in order to reconfigure the apllication in case a fault occurs. We have also discussed the space and time complexities of the developed algorithms. For more details on these algorithms the interested reader is urged to refer to [23].

4 HYBRID ALGORITHM TECHNIQUE

The idea of combining two or more different algorithms into a single hybrid algorithm was inspired by the possibility of this new algorithm performing better than any of its component algorithms individually. The result is a new class of algorithms under the umbrella of hybrid algorithms techniques(HAT). The hybrid algorithm combines the strengths of the individual algorithms so that the resulting algorithm provides a combination of the following advantages:

1. can produce better solutions;
2. and/or produce solutions in less time;
3. can tolerate software faults;
4. can effectively handle problems with larger input sizes, especially with respect to NP problems.

These advantages seem to be gained without major new disadvantages.

Figure 4.1 shows the basic idea underlying the HAT. Various algorithms co-operate towards performing a computation. At regular intervals, the results of the computation performed so far are compared by all algorithms and a good solution is distributed to all. This provides a very good mechanism for tolerating software or hardware faults because any incorrect result will be weeded out during the consensus and exchange phase.

To demonstrate the capability of HAT, we have implemented a hybrid algorithm search technique for solving combinatorial optimization problems. In order to guarantee the optimum solution for these problems, all possible solutions must be considered. Unfortunately, many of these problems fall into the class of NP-complete, and therefore the set of all possible solutions is too large to consider. Heuristics are therefore used to test only more promising subsets of the possible solutions. The existing algorithms cannot, therefore, assure the optimum solution will be found.

Several algorithms to solve combinatorial optimization problems exist. Hybridization of some of these algorithms is intended to combine the strengths of their respective heuristic techniques into a better algorithm. This new algorithm should produce solutions closer to optimal, or in less time, or both. The algorithm which produces satisfactory results in less time can also be applied to larger problems.

We expect our new hybrid algorithm search technique to be general and applicable to the majority of optimization problems. Some examples of these problems where HAST could be applied are in computer-aided design (e.g., integrated circuit or printed circuit board placement and routing), scheduling, resource allocation, test generation, integer programming and a number of graph heuristic algorithms such as coloring and partitioning. To demonstrate viability of our hypothesis of increased performance we have chosen the Traveling Salesman Problem (TSP) which is an easily defined problem in combinatorial optimization research. The problem consists of finding the shortest Hamiltonian circuit (circuit that includes every node) in a complete graph. The nodes of the graph represent cities of a map and the edges are weighted with

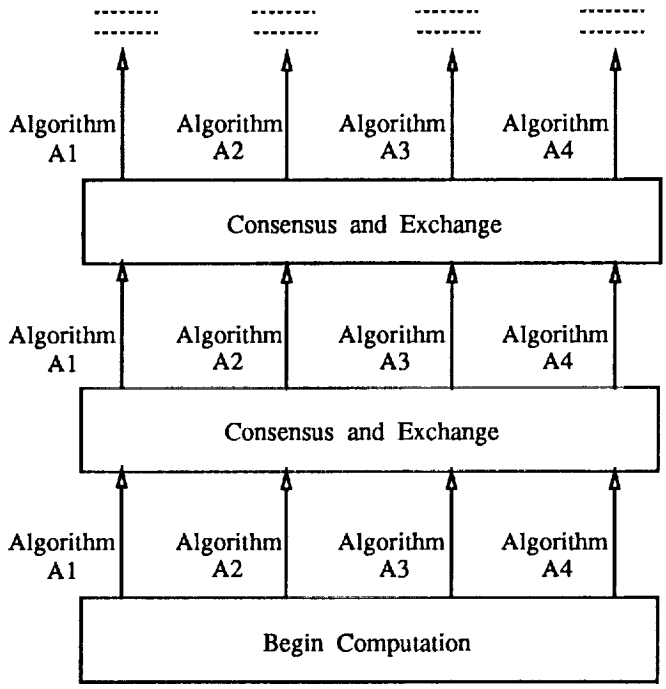


Figure 4.1: Overview of Hybrid Algorithm Technique

the distance between each pair of cities. We will define and test our algorithms with respect to the traveling salesman problem.

Our objective was to implement two different combinatorial optimization algorithms such that they may execute in parallel and exchange data periodically. The goal was to study the time efficiency and cost of mixing the simulated annealing [24] and tabu search [25] algorithms into a new parallel hybrid search algorithm as compared to these algorithms executing independently. These three search algorithms were tested on the move of the 2-opt heuristic which is based on swapping pairs of edges [26]. Experiments have been conducted on seven well known problems from the literature, namely, the 33 city, 42 city, 50 city, 57 city, 75 city, 100 city and 532 city problems. The 50 city and 75 city problems have no known optimal solutions while the others do.

4.1 Simulated Annealing/Tabu Search Hybrid (SATH)

Simulated annealing and tabu search use very different approaches to search for optimal solutions to combinatorial optimization problems. Although both of these algorithms provide good results on some problems, neither can guarantee the optimal solution will be found in real time. This, of course, leaves room for improved algorithms. We have therefore developed a hybrid algorithm in attempt to produce better performance.

SATH is a simulated annealing/tabu search hybrid algorithm, the first in a new class of easily parallelizable hybrid algorithms. SATH incorporates both

simulated annealing and tabu search as low level algorithms with a high level algorithm to mix the results from each. The idea is to execute each low level algorithm for some specified amount of time, the results of which are evaluated by the high level algorithm. The low level routines are than restarted in a more promising area of the solution space. This process is repeated as many times as is necessary or desired.

The SATH algorithm can be realized with the simulated annealing and tabu search portions implemented as subroutines. These subroutines could be executed, one after the other, followed by analysis of the results by a higher level routine. However, one of the most important features of this hybrid algorithm is the ease with which it may be executed in parallel. Each low level algorithm can be executed in parallel with a supervising process to synchronize execution and analyze results. This opens up the possibility of executing several low level algorithms in parallel, any number of which may be instances of simulated annealing or tabu search with different operating parameters. Interprocess communication is minimal and only occurs between a low level algorithm and the single high level algorithm. Speedup can therefore be linear with the number of processors as long as the number of processors does not exceed the number of low level algorithms.

4.2 Implementation of SATH

We implemented our SATH algorithm by allocating a separate process for each part of the algorithm. The basic implementation includes one main process and two child processes. When the program is executed, a main process is generated which reads in the problem definition. The main process then creates a set of child processes, one of which is a simulated annealing process, the other of which is a tabu search process. After specified time intervals, the child processes are halted and the main process compares their results. It selects a *good* solution for the child processes to continue with. A *good* solution might be the one with the least cost. In case the tour with the least cost had already been given to the child processes, passing the same tour again will result in cycling. To prevent this from happening, the tour with the next to least tour (if not previously encountered) is made the common starting point for the child processes.

Other criteria might also be applied for defining a *good* solution. In our implementation, all the processes merge at a common point in the solution space when the tour with the least cost is distributed to all of them and is used as a starting point for the next iteration. Several other approaches might be considered, one of them being pseudorandomization. In this case, each process starts off with a pseudorandom tour after the information has been exchanged. This can be achieved by maintaining a history of the search space visited be each process in the previous iterations. Thus the new starting tours after the information exchange will be composed from previous history stored in the long term memory and information about the covered search space.

Implemented in this fashion the SATH algorithm can be executed on a single processor or on multiple processors with very little effort. The algorithm is also expandable by adding additional simulated annealing and tabu search processes executing with different search parameters. The algorithm can be expanded in this way until there is a process for every available processor. To execute the SATH algorithm on a single processor requires that only one

processor is available to execute the processes. In addition, synchronization was added to assure one process executes to completion before the next one begins.

In our SATH algorithm each simulated annealing process executes with a different annealing schedule. The schedules are chosen as in the accelerated simulated annealing algorithm described in [26]. When the SATH algorithm had multiple tabu search processes, each process had a different tabu condition and a corresponding tabu list size to distribute the search in the solution space.

4.3 Experimental Results

Our experiments with the traveling salesman problem have illustrated the advantages of using a hybrid search technique based on mixing simulated annealing and tabu search algorithms. The hybrid algorithm performs very well for all of the investigated problems, namely 33, 42, 50, 57, 75, 100 and 532 city problems. It holds considerable potential for reducing execution time for solving NP-complete problems and at the same time improving the quality of the solution. For a detailed description of the results and performance of our approach, we suggest the reader to refer to [26]–[28]. In our opinion, the hybrid algorithm is very well suited for other problems as well, not necessarily search techniques. With the advent of parallel processing in the computing environment, it becomes especially attractive to exploit the inherent parallelism in the proposed algorithm. A major advantage of the proposed approach is the ability to tolerate software faults due to multiple algorithm implementations. In addition, hardware faults can be tolerated in multiprocessor implementation of the HAT.

5 CONCLUSIONS

This research has made contributions towards the management of redundancy in computer systems and provided guidelines for the development of NASA's fault-tolerant distributed systems. Our view of fault tolerance consisting of four layers will facilitate the design of fault-tolerant distributed systems. This four-layered view provides a comprehensive approach for achieving reliable computations. Such a global picture is indispensable for obtaining effective redundancy management, especially for distributed systems. A starting point in fault-tolerant system implementation is solving the issue of synchronization. Once synchronization has been incorporated, the problem of reliable broadcast can be addressed. This can be followed by achieving consensus for identifying faulty computation units. Finally, reconfiguration and recovery from faults are tackled. Our research has been largely concentrated on fault recovery and reconfiguration mechanisms.

Our work has laid a theoretical foundation for redundancy management by efficient reconfiguration methods and algorithmic diversity. It has defined various parameters that are of interest to the user for managing space and time redundancy to achieve fault tolerance in a system. Algorithms have been developed to optimize the resources for embedding of computational graphs of tasks in the system architecture and reconfiguration of these tasks after a failure has occurred. The computational structure represented by a path and the complete binary tree have been considered and the mesh and hypercube architectures have been targeted for their embeddings. The innovative concept of Hybrid Algorithm Technique has been introduced. This new technique provides a mechanism for obtaining fault tolerance while exhibiting improved performance.

One of the problems which we are currently investigating is that of minimally biconnected networks. Usually, for a distributed system, the network does not have redundant paths for all pairs of nodes. Hence the failure of a communication channel might disconnect the network. For the sake of fault tolerance it is desirable that the existing network be modified to increase the connectivity. The aim is to achieve this goal while minimizing additional parameters such as the number of extra channels or the length of the cycle and/or maximizing the use of existing resources.

Our current research has been directed towards introducing fault tolerance in real-time systems. These fault-tolerant real-time systems, called *responsive systems* [29], are required for very critical applications, such as NASA's future Space Station. Redundancy Management to obtain fault tolerance in such system is a challenging task due to the additional constraints of real-time and criticality of application. Our approach is towards a comprehensive design of such systems including specification, modeling and design for redundancy management and recoverability.

Our research in 1989/90 resulted in nine publications, three research reports and two book chapters. Since our research contributed to formalization and quantification of efficient recovery methods, we anticipate that it will facilitate redundancy management in NASA's distributed computing systems.

References

[1] Cristian, F., "Probabilistic clock synchronization", *Proceedings of the Eighth International Conference on Distributed Computing*, vol. 3, no. 3, 1989, pp. 146-158.

[2] Cristian, F., H. Aghili and R. Strong, "Atomic broadcast: from simple message diffusion to Byzantine Agreement", Research Report RJ 5244 (54244), IBM, July 1986.

[3] Babaoglu, O., P. Stephenson and R. Drumond, "Reliable broadcasts and communication models: tradeoffs and lower bounds", *Distributed Computing*, No. 2, 1988, pp. 177-189.

[4] Birman, K. and T. Joseph, "Reliable communication in the presence of failures", *ACM Transactions on Computer Systems*, vol. 5, no. 1, February 1987, pp. 47-76.

[5] Chang, J. and N. Maxemchuck, "Reliable broadcast protocols", *ACM Transactions on Computer Systems*, vol. 2, no. 3, pp. 251-273, 1984.

[6] Nett, E., K.Großpietsch, A. Jungblut, J. Kaiser, R. Kröger, W. Lux, M. Speicher and H. Winnebeck, "PROFEMO design and implementation of a fault tolerant distributed system architecture", Technical Report No. 100, GMD-Studien, June 1985.

[7] Cristian, F., "Agreeing on who is present and who is absent in a synchronous distributed system", *Proceedings of the Fault-tolerant Computing Symposium*, June 1988, pp. 206-211.

[8] Barborak, M., M. Malek and A. Dahbura, "The consensus problem in fault-tolerant computing", Technical Report, Department of Electrical and Computer Engineering, The University of Texas at Austin, February 1991.

[9] Malek, M., "A comparison connection assignment for diagnosis of multiprocessor systems", *Seventh Computer Architecture Symposium*, May 1980, pp. 31-36.

[10] Bianchini, R. Jr., K. Goodwin and D. Nydick, "Practical application and implementation of distributed system-level diagnosis theory", *Proceedings of the 20th Fault-Tolerant Computing Symposium*, June 1990.

[11] Cristian, F., B. Dancey and J. Dehn, "Fault tolerance in the advanced automation systems", *Proceedings of the 20th Fault-Tolerant Computing Symposium*, 1990, pp. 6-17.

[12] Siewiorek, D. and R. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, 1982.

[13] Randell, B., "System structure for fault tolerance", *IEEE Transactions on Software Engineering*, vol. SE-1, June 1975, pp. 220-232.

[14] Harary, F. and M. Malek, "Fault recovery in multiprocessor systems: a graph theoretic approach", Technical Report, Department of Electrical and Computer Engineering, The University of Texas at Austin, 1987.

[15] Harary, F. and M. Malek, "Quantifying fault recovery in multiprocessor systems", *Computers and Mathematics with Applications*, Pergamon Press, 1989.

[16] Malek, M. and Y. Choi, "A fault-tolerant FFT processor", *IEEE Transactions on Computers*, vol. C-37, no. 5, May 1988, pp. 617-621.

[17] Malek, M. and K. Yau, "The resiliency triple in multiprocessor systems", *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988, pp. 351-358.

[18] Youn, H. and A. Singh, "On area efficient and fault-tolerant tree embedding in VLSI", *Proceedings of the 1987 International Conference on Parallel Processing*, August 1987, pp. 170-177.

[19] Gordon, D., "Efficient embeddings of binary trees in VLSI arrays", *IEEE Transactions on Computers*, vol. C-36, no. 9, September 1987, pp. 1009-1018.

[20] Mead, C. and L. Conway, *Introduction to VLSI Systems*, Reading, Mass., Addison Wesley, 1980.

[21] Mead, C. and M. Rem, "Cost and performance of VLSI computing structures", *IEEE Journal of Solid State Circuit*, vol. SC-14, no. 2, April 1979.

[22] Deshpande, S. and R. Jenevein, "Scalability of a binary tree on a hypercube", *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986, pp. 661-668.

[23] Yau, K., "The Analysis and Optimization of Fault Tolerance in Multiprocessor Systems: A Graph Theoretic Approach", Ph.D. Dissertation, Department of Electrical and Computer Engineering, The University of Texas at Austin, May 1989.

[24] Kirkpatrick, S., C. Gelatt and M. Vechi, "Optimization by simulated annealing", *Science*, vol. 220, no. 4598, May 13, 1983.

[25] Glover, F., "Tabu search methods in artificial intelligence and operations research", *ORSA Artificial Intelligence Newsletter*, vol. 1, 1987.

[26] Malek, M., M. Guruswamy, H. Owens, and M. Pandya, "Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem", *Annals of Operations Research*, 21, pp. 59-84, 1989.

[27] Malek, M., M. Guruswamy, H. Owens, and M. Pandya, "The hybrid algorithm technique", Technical Report TR-89-06, Department of Computer Sciences, The university of Texas at Austin, March 1889.

[28] Mourad, A. and M. Malek, "Fault-tolerant parallel algorithm design", Department of Electrical and Computer Engineering, The University of Texas at Austin, August 1989.

[29] Malek, M., "Responsive systems: a challenge for the nineties", *Proceedings of Euromicro 90, Sixteenth Symposium on Microprocessing and Microprogramming*, Keynote address, Amsterdam, The Netherlands, North-Holland, Microprocessing and Microprogramming 30, pp. 9-16, August 1990.